15

CLaSSiC Project
Manuscript CLaSSiC-87-17

February 1987

Copy Elimination with Abstract Interpretation

K. Gopinath
John L. Hennessy

Center for Large Scale Scientific Computation
Building 460, Room 313
Stanford University
Stanford, California 94305

-A-

Copy elimination with Abstract Interpretation*

K.Gopinath and John L.Hennessy
Computer Systems Laboratory
Stanford University

Abstract

Copy elimination is an important optimization for implementing functional languages. Though it is related to the problem of copy propagation that has been considered in many compilers and also to storage compaction, the term is used in a more general context where structured values can be updated and the computation tree can be reordered. Because of these two additional possibilities, copy elimination is a hard problem, being undecidable in general.

We propose an optimization approach based on abstract interpretation which uses fixpoint iteration for computing *address expressions*. These address expressions supply the final target for a computation, eliminating the need to copy values through intermediate results. Our work is in the context of a single assignment language called SAL. Our implementation has an operational model for computing address expressions by using reduction rules. Using this, we show that copies present in divide and conquer algorithms like *bitonic sort* and *quicksort* can be removed. We evaluate the effectiveness of these optimizations, showing that in many cases, we can come close to the efficiency of an imperative language. We present some data on optimising some small but tough benchmarks.

* This work was supported in part by the Office of Naval Research Contract N00014-82-K-0335.

Table of Contents

Copy elimination with Abstract Interpretation

K.Gopinath and John L.Hennessy
Computer Systems Laboratory
Stanford University

1. Introduction

Copy elimination is an important optimization for implementing functional languages. Though it is related to the problem of copy propagation that has been considered in many compilers [1], the term is used in a more general context where structured values can be updated and the computation tree can be reordered. Because of these two additional possibilities, copy elimination is a hard problem, being undecidable in general. Copy propagation is adequate in imperative languages at intermediate representation level since the programmer takes responsibility for avoiding any unnecessary copies in the source language, such copies being considered a reflection of poor programming. In functional programs, however, the lack of variables in the language requires that a value incrementally different from some structure be expressed as a new structure. This may involve a copy depending on the program semantics and the implementation. Copy elimination also differs significantly from storage compaction where the issue is the computation of a program in the smallest amount of storage. An effective storage compaction scheme might avoid all the copies at the same time but likely to be expensive computationally, since attempt is made to assign even unrelated names into the same storage space, partially or fully, as long as they have non-overlapping live ranges. The latter strategy is useful in the case of limited resources like registers but minimizing main memory is of less value.

In this paper, we will use the technique of abstract interpretation for copy elimination. Abstract interpretation is a technique that was pioneered by Cousot and Cousot [5] for deriving properties of programs. Using this approach, Alan Mycroft [12] considered the problem of detecting when a call-by-need argument can be turned into a call-by-value argument in the interests of efficiency. Hudak [9] in his recent work has used the technique successfully to detect updates that can be done in-place by reference counting. We will use abstract interpretation to eliminate copies also. This approach involves fixpoint iteration for computing *address expressions* in the presence of recursive functions. These address expressions can be used to compute the final target address for a computation, eliminating intermediate copies. The insights gained by this approach will be used to compute address expressions using *reduction rules* for a large subset of a single assignment language called SAL. We will give a brief description of SAL after describing how abstract interpretation can be used to eliminate copies. This approach has been used to remove copies in divide and conquer problems like quicksort and bitonic sort and we report times which are very close to those for imperative languages like Pascal on a set of small, but tough benchmarks.

2. Abstract interpretation

Let the standard denotation of a function definition be $\ll f \gg : D \rightarrow R$. Abstraction functions are then defined to "simplify" the domains of D and R. If two such functions *Abs1*: $D \rightarrow A1$ and *Ab2*: $R \rightarrow A2$ are chosen properly, the function $f : D \rightarrow R$ induces another interpretation $\ll$ *abs-f* $\gg : A1 \rightarrow A2$ which can be used to answer some of the properties of f. This is the central idea of abstract interpretation. A good example is the function *type* which abstracts just the type aspects of a function. There has been considerable literature on using abstract interpretation for computing strictness of parameters.

We use the abstraction ***Addr*** for both *Abs1* and *Abs2* in our application with both *A1* and *A2* being the domain of address expressions. The abstraction ***Addr*** maps names into their symbolic addresses and expressions into address expressions. Analogous to type expressions, address expressions describe the address of an expression in terms of the parameters. These are similar to the *effect-declarations* of Schwarz [13]. The mapping *abs-f* gives the properties of sharing that is possible between arguments and the result of a function.

3. Assumptions

We assume a simple language similar to the one considered by Hudak [9]:

Program is a set of recursive function definitions (without higher-order functions)

$f_i := \lambda x_1,\ldots,x_n\ body_i$

with main program being a call of f_1 with zero arguments.

The predefined functions p_i are as follows:

- standard arithmetic, boolean and array selector operations
- conditional: **if** p **then** c **else** a abbreviated as **if**(p,c,a)
- create array function: **mka**(*bounds*, h) where *bounds* is an integer range and h is some function which maps *bounds* to some image set.
- update function: **upd**(A,i,v) where A is an array, i is a legal index into A and v is of the same type as A's elements and its value is same as the array A except at the ith position where it is v.
- sequence function: **seq**(A,l,m) where A is an array and l and m are legal indices into A and its meaning is same as the subarray of A starting at index l and ending at m. This will be abbreviated to $A[l..m]$

- catenate function: **cat**(A,B) where A and B are sequences. For purposes of optimization and the non-standard semantics that will be shortly developed, it has the following meaning: if $A=a[l..m]$ and $B=a[n..p]$ then **cat**(A,B) is $a[l..p]$ if $m+1=n$; otherwise, a newly created array $c[1..m\text{-}l+1+p\text{-}n+1]$ where $c[1..m\text{-}l+1]=a[l..m]$ and $c[m\text{-}l+2..m\text{-}l+1+p\text{-}n+1]=a[n..p]$

We adopt the convention that an identifier in bold italics represents the address expression for the corresponding identifier written in italics. We adopt the same convention for representing terms in standard and non-standard semantics.

4. A Denotational model

We use a denotational model for the simple language to show that fixpoint computation of address expressions is possible and that they terminate.

4.1. Preliminaries

We adopt a slightly modified version of the notation in [9]. Double angle brackets are used to surround syntactic objects, as in $\boldsymbol{E}\langle\!\langle exp\rangle\!\rangle$. A new environment is created by $[e_1/x_1,\ldots e_n/x_n]$ which is abbreviated to $[e_i/x_i]$ when the subscript bounds are clear from context. The notation $A^* \rightarrow B$ denotes the domain $B+(A\rightarrow B)+(A\rightarrow A\rightarrow B)+\cdots$ We also assume that all domains are "lifted" as necessary, *i.e.*, they are provided with an unique least element. We will refer to each of them by $\perp$ instead of different ones for each domain.

4.2. Standard semantics

Let D be some suitable domain of basic values. For the standard semantics, we have the following domains:

$c,p \in Con$ (constants including primitive functions)
$x \in Bv$ (bound variables)
$f \in Fv$ (function variables)
$body,e \in Exp$ (expressions)
where $e ::= c \mid x \mid p(e_1 \ldots e_n) \mid f(e_1 \ldots e_n)$
$pr \in Prog$ (programs)
where $pr ::=$

$$\begin{aligned}\{\ & f_1(x_{11}\ldots x_{1k_1}) = body_1\\ & f_2(x_{21}\ldots x_{2k_2}) = body_2\\ & \ldots\\ & f_n(x_{n1}\ldots x_{nk_n}) = body_n\}\end{aligned}$$

Define two environments bve and fve, for bound and function variables, respectively. Let E_p be the semantic function for giving meaning to programs and E and E_k correspondingly for expressions and constants. The semantic algebra is as follows:

$fve \in Fve = Fv \rightarrow D^* \rightarrow D$
$bve \in Bve = Bv \rightarrow D$
$E_p : Prog \rightarrow Fve$
$E : Exp \rightarrow Fve \rightarrow Bve \rightarrow D$
$K : Con \rightarrow Exp^* \rightarrow Fve \rightarrow Bve \rightarrow D$ (assumed given)

The semantic equations are as follows:

$E_p \ll \{f_i(x_1,\ldots,x_m)=body_i | i:1..n\} \gg = fve$ **whererec** $fve = [\, E \ll body_i \gg fve \;\; bve \,/\, f_i]$

$E \ll c \gg fve\; bve = K \ll c \gg fve\; bve$
$E \ll x \gg fve\; bve = bve \ll x \gg$
$E \ll p(e_1,\ldots,e_n) \gg fve\; bve = K \ll p \gg (E \ll e_1 \gg fve\; bve), \ldots ,(E \ll e_n \gg fve\; bve)$
$E \ll f_i(e_1,\ldots,e_n) \gg fve\; bve = fve \ll f_i \gg (E \ll e_1 \gg fve\; bve), \ldots ,(E \ll e_n \gg fve\; bve)$

4.3. Non-standard semantics

Given a program, the non-standard semantics derives a set of recursive equations for the address expressions of functions which can then be computed by fixpoint iteration.

Let G be a set of names for anonymous arrays created by **mka**, **cat**, **upd** array operations and also arrays created by **if** expressions so that there is a 1-1 correspondence between occurrences of these array creating operations and the set G. Let each of these operations be labelled with an integer value so that the symbolic name for the i th occurrence of any of these operations considered together is g_i. Also let

$P(A)$ be the powerset of A.
$\boldsymbol{x} = \boldsymbol{Addr}(x)$ for each $x \in Bv \cup G$;
$\boldsymbol{X} = \{\boldsymbol{x} | x \in Bv \cup G\}$
$\boldsymbol{G} = \{\boldsymbol{x} | x \in G\}$
RN_x be the subscript range of an array x; assume that $\boldsymbol{x}[RN_x]$ is always rewritten as $\boldsymbol{x}$
$\boldsymbol{A} = \{\boldsymbol{x}[l..m] | \text{array } x \in Bv \cup G;\ l,m \in RN_x\} - \boldsymbol{X}$; so that $\boldsymbol{A}$ and $\boldsymbol{X}$ are disjoint.
$AddrExp = \{\bot, \top\} \cup P(\boldsymbol{X}) \cup P(\boldsymbol{A})$

$AddrExp$ is a pointed cpo with the following partial ordering:

$\bot <= a <= \top,\ a \in AddrExp$
$a <= b$ if $a \subseteq b$

$\bot$ and $\top$ are the empty and universe set respectively. We define two monotonic operators μ and ν (used for defining the semantics of **if** and **cat** respectively) over the domain $AddrExp$ as follows:

$\mu(a,b,g)$:

$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa \in \mathcal{P}(\mathbf{X})$ and $bb \in \mathcal{P}(\mathbf{A})$ & vice-versa
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa \in \mathcal{P}(\mathbf{G})$ and $bb \in \mathcal{P}(\mathbf{X\text{-}G})$ & vice-versa
$=g \quad \exists aa \subseteq a \ \&\ bb \subseteq b$ such that $aa=\{x[l..m]\}$, $bb=\{x[n..p]\}$, $(l \neq n$ or $m \neq p)$
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa=\{x[l..m]\}$, $bb=\{y[n..p]\}$, $x \neq y \ \&\ x,y \in G$
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $\mu(aa, bb, g) \in G$
$=a \sqcup b$ otherwise

$\nu(a,b,g)$

$=\{x[l..p]\}$ if $a=\{x[l..m]\}$ and $b=\{x[n..p]\}$ and $n=m+1$.
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa \in \mathcal{P}(\mathbf{X})$ and $bb \in \mathcal{P}(\mathbf{A})$ & vice-versa
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa \in \mathcal{P}(\mathbf{G})$ and $bb \in \mathcal{P}(\mathbf{X\text{-}G})$ & vice-versa
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa=\{x[l..m]\}$, $bb=\{x[n..p]\}$ & $n \neq m+1$
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $\nu(aa, bb, g) \in G$
$=g \quad \exists\, aa \subseteq a \ \&\ bb \subseteq b$ such that $aa=\{x[l..m]\}$, $bb=\{y[n..p]\}$, $x \neq y \ \&\ x,y \in G$

$\perp$ is the bottom element for the cpo, signifying null information. If the two operands are incompatible, the symbolic address for the anonymous array (passed as g) is returned.

We have the following new semantic algebra:

$Loc=AddrExp$
$Bve=Bv \rightarrow Loc$
$St=Loc \rightarrow D$
$\mathbf{fve} \in \mathbf{Fve}=\mathbf{Fv} \rightarrow Loc^* \rightarrow St \rightarrow (Loc \times St)$
$EE_p\!:Prog \rightarrow \mathbf{Fve}$
$EE\!:Exp \rightarrow \mathbf{Fve} \rightarrow Loc$
$KK\!:Con \rightarrow Exp^* \rightarrow \mathbf{Fve} \rightarrow Loc$

The **semantic functions** for the abstraction have some unusual features. They depend on the standard semantics also. Hence, a product construction is needed to describe the semantics. For simplicity of presentation, we just present the non-standard semantics, the product construction being understood. We will discuss how to eliminate this dependence on the standard semantics since discovering properties at compile time for optimization purposes is not possible otherwise. In addition, we have the problem of non-termination in the standard semantics. We use the notation lab_e for the label corresponding to expression e.

$EE_p \ll \{f_i(x_1,\ldots,x_m)=body_i|i{:}1..n\} \gg = \mathbf{fve\ whererec\ fve} = [EE \ll body_i \gg \mathbf{fve} / f_i]$

$EE \ll c \gg \mathbf{fve} = \top$
$EE \ll x \gg \mathbf{fve} = x$
$KK \ll arith\text{-}bool \gg e^*\ \mathbf{fve} = \top$

$EE \ll f(e_1,\ldots,e_n) \gg \mathbf{fve} = \mathbf{fve} \ll f \gg (EE \ll e_1 \gg \mathbf{fve}) \ldots (EE \ll e_n \gg \mathbf{fve})$

```
KK ≪ if ≫ cond conseq alt fve =
  let
   $t_1$:=EE ≪ conseq ≫ fve
   $t_2$:=EE ≪ alt ≫ fve
  in
   $\mu(t_1,t_2,g_{lab_{if}})$
  end

KK ≪ cat ≫ A[l..m] B[n..p] fve =
  let
   ll:=E ≪ l ≫ fve bve
   mm:=E ≪ m ≫ fve bve
   nn:=E ≪ n ≫ fve bve
   pp:=E ≪ p ≫ fve bve
   $t_1$:=(EE ≪ A ≫ fve)[ll..mm]
   $t_2$:=(EE ≪ B ≫ fve)[nn..pp]
  in
   $\nu(t_1,t_2,g_{lab_{cat}})$
  end

KK ≪ mka ≫ bounds h fve = $g_{lab_{mka}}$

KK ≪ upd ≫ A i v fve =
 let
  t:=EE ≪ A ≫ fve
 in
  if A is not live then t else $g_{lab_{upd}}$
 end
 (*liveness can be deduced by using Hudak's denotational semantics for reference
   counting by using a product construction; see discussion below*)
```

4.4. Discussion

μ or ν cannot be simplified to g if the subsets aa or bb are both from $P(X)$ or $P(A)$. Consider the following function:

function $Q(A,B{:}arr){:}arr$=**if** *cond* **then** A **else** B

$Q=\lambda A\ B.\ \{A,\ B\}$

Since Q can be called with arbitrary parameters A and B, it is not possible to determine the address expression of the function body in a form simpler than $\{A,\ B\}$. If this is simplified to g, this may give us pessimistic information about what can be shared in case Q is called with actual parameters such that $A{=}{=}B$. It might be thought that the problem can be eliminated if functions are in-lined but we see a similar problem in the following recursive example (which cannot be in-lined).

```
f(a,b,c)=
  if cond1 then cat(a,b)
  elsif cond2 f(upd(a,i,v1), b, h1(c))
  else f(a, upd(b,j,v2), h2(c))
```

f=$\lambda abc.\ \nu(a,b,g_{lab_{\mathbf{cat}}}) \cup f(a,b,h1(c)) \cup f(a,b,h2(c))$

The expression in the *then* part cannot be simplified in the abstract interpretation to g. If f is called in the form: $f(A[1..m], B[m+1..n],c)$ with $A{=}B$, then we get pessimistic answers. The simplification to g results in the failure to propagate information across function boundaries. This is not just a theoretical possibility; the bitonic sort program has this property. A similar problem exists in strictness analysis [8].

We need to make some small changes to Hudak's semantics for reference counting: If **cat**($A[l..m]$, $B[n..p]$) can be updated in-place because $A{=}B$ and $n{=}m{+}1$, the reference count for A does not decrease. If this is not the case, the reference counts for both A and B are reduced by one, just as in the standard semantics.

4.5. Elimination of the standard semantics

We next discuss how to eliminate the standard semantics from the equations. Consider the equation for **cat**. One of the conditions for computing ν is $(n{=}m{+}1)$. Instead of evaluating this by using the standard semantics, it can be checked by symbolic analysis if the syntactic expressions for n and m are of a particularly simple nature, namely induction variables. The condition $n{=}m{+}1$ can be checked by matching subtrees of n and m. If the expressions for n and m do not satisfy the simple syntactic criteria, the value **false** is returned. The abstraction, in this case, errs on the safe side. Also note that computation of ll and pp is not necessary. Similarly, the check $(l \neq n$ or $m \neq p)$ for computing μ can be carried out symbolically. The new semantic equation for **cat** is as follows:

```
$KK \ll \mathbf{cat} \gg A[l..m]\ B[n..p]\ \mathbf{fve}$ =
  let
    $\mathbf{t}_1 := EE \ll A \gg \mathbf{fve}[\ll l \gg .. \ll m \gg]$
    $\mathbf{t}_2 := EE \ll B \gg \mathbf{fve}[\ll n \gg .. \ll p \gg]$
  in
    $\nu(\mathbf{t}_1, \mathbf{t}_2, g_{lab_{\mathrm{cat}}})$
  end
```

4.6. Some results

Theorem 1: For any finite program $pr \in Prog$ (with bounded arrays), the fixpoints corresponding to $f \in Fv$ are computable.

Proof: *AddrExp* is a finite cpo. Also, *EE*, *KK* are constructed from monotonic operations. Hence ***fve*** can be effectively computed by fixpoint iteration starting with the bottom element and iterating till the least upper bound is reached.

Theorem 2: If the fixpoint solution of $f_i \in Fv$ is some x, then the body of the function can be targeted to x and x can be updated in-place in the body of f_i to give the result of the function.

Proof:The proof is by structural induction on $body_i$. Since f_i is a fixpoint, $f_i(x)=F(f_i)(x)= x$. Consider the following cases for $body_i$ in the semantic functions EE and KK:

x: If $body_i = x$, the result is immediate.

$h(e_1,...,e_n)$: f_i has x as fixpoint only if the abstraction of h maps some argument e_i into x. By induction hypothesis, h's result can be obtained by the update of this argument. Hence f_i's result can also be obtained from the update of the same argument.

if(*cond*,*conseq*,*alt*): f_i has x as a fixpoint only if both arms of the conditional map to the same address expression. From the induction hypothesis, it follows that the result of the function is given by the update of the parameter x if *cond* is true and also by x when *cond* is false. Hence the result is given by update of x.

cat($A[l..m]$,$B[n..p]$): f_i has x as fixpoint only if both A and B are mapped to x and $n=m+1$ with l and p as the lower and upper bounds of x. Hence the result of the function is given by update of x if **cat** function is in-place for this condition.

upd(A,i,v): Again, f_i has x as fixpoint only if A is mapped to x. If A is not live then x can be updated to give the result of f_i.

mka: This case cannot give rise to x as a fixpoint, hence vacuously true.

This completes the theorem's proof by structural induction.

Theoiem 3: The abstraction is safe.

Proof: The proof is by structural induction which is omitted.

4.7. An example

To illustrate the non-standard semantics given above, consider the program for reversing the elements of an array:

```
function swap2(A:arr;Ai,Aj:arrelem;i:integer;j:integer):arr=
         upd(upd(A,i,Aj),j,Ai)

function rev(Ap, Aq:arrelem, A:arr; i, n:integer)-
         if n=1 or i=n/2 then A
         else rev(A[n-i+2], A[i-1], swap2(A, Ap, Aq, n-i+1, i), i-1, n)
```

If *swap2* is to be implemented in-place two additional paramters are needed because of the lack of assignments. When the abstraction ***Addr*** is used, the new interpretation of the two equations for *swap2* and *rev* by the non-standard semantics is as follows:

function ***swap2***(***A***:*arr*;***Ai***, ***Aj***:*arrelem*;*i*:**integer**;*j*:**integer**):*arr*= $\{A\}$

function ***rev***(***Ap***, ***Aq***:*arrelem*;***A***:*arr*;*i*, *n*:*integer*):*arr*=
$\{A\} \cup \{rev(A[n\text{-}i+2], A[i\text{-}1], A, i\text{-}1, n)\}$

Since *swap2* occurs as a leaf in the call-graph, it is advantageous to find the interpretation of *swap2* before *rev*. The new interpretation of *swap2* is computed to be just ***A*** by using Hudak's semantics for reference counting whereas the interpretation for *rev* has to be found by fixpoint iteration as follows:

$rev_0 := \perp$
$rev_1 := \{A\} \cup \perp := \{A\}$
$rev_2 := \{A\} \cup \{A\} := \{A\}$

Hence, abstract interpretation of *rev* under mapping ***Addr*** is just ***A***. Hence the result of the function can be given by updates on the parameter *A* from Theorem 2.

4.8. Applicability to other functional languages

We discuss briefly the applicability of the approach we have taken for copy elimination in the context of features that are not present in the simple language considered.

Call-by-need/call-by-value/lazy evaluation: A perusal of the non-standard semantics shows that these evaluation mechanisms make their effects felt through the computation of liveness of names. Hudak and Bloss [2] have considered the problem of computing the order of evaluation of subexpressions in the context of call-by-need parameter evaluation mechanism and this can be used for determining liveness. Lazy evaluation, which differs from call-by-need in evaluating an expression more than once if needed, presents considerable difficulties. If infinite structures are present, the structure is live and lazy evaluation is needed. We, therefore, need analysis to detect if a structure necessarily has to be evaluated by lazy evaluation and analysis to convert lazy evaluation to call-by-need or other forms of evaluation. Wadler [14] considers some of these issues for a limited class of situations arising in practice.

Higher-order functions: Higher order functions could be handled if Hudak's semantics can be extended to higher-order functions (which has not been done yet). Even with this assumption, termination can no longer be guaranteed. To see why, consider the simple language extended with higher order functions. The new domains are as follows:

$c,p \in Con$	(constants including primitive functions)
$f_i, lambda \in Fv$	(function variables with bodies and anonymous lambda expressions)
$x,h \in Bv$	(bound variables including function parameters)
$body,e \in Exp$	(expressions)

where $e ::= c \mid x \mid p(e_1 \ldots e_n) \mid f(e_1 \ldots e_n) \mid \lambda x.e$

bv includes formal parameters which are functions whereas *fv* also includes anonymous lambda abstractions. We need additional semantic rules to take care of lambda abstractions:

swap2(A:arr;Ai,Aj:arrelem;i:integer;j:integer):arr= {A}

$EE \ll f_i \gg fve = f_i$
$EE \ll \lambda x.e \gg fve = \lambda x.\ EE \ll e \gg fve$
$EE \ll f(e_1,\ldots,e_n) \gg fve = (\ EE \ll f \gg fve)\ (EE \ll e_1 \gg fve\ ) \ldots (\ EE \ll e_n \gg fve\ )$

The above semantic rules cause the evaluation of the function part of an application since there could be anonymous lambda expressions.

Let us see why there can be non-termination. Consider the following program:

```
f(h,a,b,c)=
  if cond then h(h,h(h,a,b,c),b,c)
  else f(h,upd(a,i,v1), upd(b,j,v2), h(c))
```

Using the non-standard semantics, we have the following fixpoint iteration:

$f_0 := \perp$
$f_1 := \{h(h,h(h,a,b,c),b,c)\} \cup \perp := \{h(h,h(h,a,b,c),\ b,c)\}$
$f_2 := \{h(h,h(h,a,b,c),\ b,c)\} \cup \{h(h,h(h,a,b,c),b,c)\}] := \{h(h,h(h,a,b,c),\ b,c)\}$

Consider now the self-application of f by substituting f itself for h. The fixpoint is not defined because of the non-terminating computation involved.

5. Copy elimination in SAL

SAL is a single assignment language defined at Stanford [3] providing iteration, parametric types and streams with scoping mechanisms similar to Algol languages.

We briefly describe some of the constructs of the SAL language:

- **simple let-expression**: allows for bindings of values to names.
- **array former**: specifies an order-independent iteration (*for all*) which produces an array result: **for** *i:l..m* **be** *exp* which is similar to **mka**(*bounds*, *h*).
- **redefining let-expressions**: implements iteration with loop dependencies; the semantics are very close to what is found in VAL and SISAL. The iterative statements have a set of initial value definitions, a corresponding set of redefinitions used to define new values on every iteration, a loop control, and a result value defined in terms of the final values of the names in the redefinitions. To be able to specify the redefinitions without any implied sequencing, the keyword **old** is used to refer to the value of a name in the previous iteration. Every redefining let-expression can be rewritten using only tail-recursive functions.
- **constructor**: creates structured values, similar to aggregates in Ada. Constructors are often used with a left hand side consisting of multiple **components** as in ML and Algol68
- **new array**: creates a value which matches the argument array but with one component value changed: *A:i- v* which is same as **upd**(*A*,*i*,*v*)
- **record field selector array element selector**: selector operations in records and arrays
- **insert expression**: the reduction operator similar to the operator in FP

The atomic datatypes are integer, boolean and reals. The structures present are arrays, records, discriminated unions and streams. For simplicity of presentation, we deal mainly with arrays but many of the issues discussed typically arise in records and discriminated unions also.

5.1. Computing Address expressions of SAL constructs

The address expression for a simple let-expression is easy to state but redefining let-expression are difficult to handle in the presence of optimizations (sharing between new and old values which requires a form of dependency analysis considered in vectorizing compilers) needed to implement it efficiently. This makes computation of the address expression using denotational semantics difficult. We will, therefore, develop an alternate approach to tackle this problem using reduction rules.

Constructors enable more than one value to be returned. This is especially useful for implementing non-locals by returning updated non-locals along with the result of the function value. We need an additional semantic equation:

$$EE\ll[e_1,\ldots,e_n]\gg fve=[EE\ll e_1\gg fve,\ldots,EE\ll e_n\gg fve]$$

The domain now has a more complex cross product structure.

5.2. Computing address expressions using reduction rules

A *reduction rule* describes how the address of an arbitrary expression can be reduced into an address of a simpler expression. Each reduction rule has two components, a *cond* and a *reduction rule*, the *reduction* being performed only if *cond* is true. Define *terminal expression* as the irreducible term of an expression.

A set of reduction rules is presented in Figures 5-1, 5-2. These rules are not complete and do not take into account many subtle details that are important in optimization. Due to lack of space, we also do not consider dependent iterations, though we consider examples which have these constructs.

5.3. Examples

Consider the following program:

```
    --initialises all the elements of an array.
    type arr=array[1..m] of integer;
    function init(a:arr;x,n:integer).arr=if n=0 then a else init(a:n->x,x,n-1)
```

Let us compute the address expression of the function body.

```
    Addr(if n=0 then a else init(a n->x,x,n-1))
    =>if n=0 then Addr(a) else Addr(init(a:n->x,x,n-1))
```

Here, we have used the following slightly different rule for conditional, for purposes of explanation:

```
    Addr(if condition then X else Y)      if condition then Addr(X) else Addr(Y)
```

No more reductions are possible unless we know that *init* can update its first argument to give the result

1. ***Addr*(if *condition* then *X* else *Y*)=>*Addr*(*X*)**, if ***Addr*(*X*)=*Addr*(*Y*)**

2. Let $f=\lambda(a,b,...)$ *exp* and $f=\lambda.(a,b,...)$ *a-exp*
   Then ***Addr*(*f*(*x*,*y*,...))=>*f*(*a-exp*)[*Addr*(*x*)/*a*,*Addr*(*y*)/*b*,...]**

3. ***Addr*(for *i*:*l*..*h* be *A*[*i*+*c*])=>*Addr*(*A*)[*l*+*c*..*h*+*c*]**, *c* a constant

4. ***Addr*(for *i*:1..*n* be if *i*=1 then *c* else *A*[*i*-1])=> *Addr*(*A*)[1..*n*]**, if *A* is not live and if loop is executed in the reverse order(from *n* to 1)[1]

5. ***Addr*([exp_1,exp_2,...,exp_n])=> [*Addr*(exp_1),*Addr*(exp_2),...,*Addr*(exp_n)]**

6. ***Addr*(cat(*A*[*l*..*h*],*B*[*m*..*n*]))=>*Addr*(*A*)[*l*..*n*]**, if *m*=*h*+1 and ***Addr*(*A*)=*Addr*(*B*)**

7. ***Addr*(*A*:*i*->*v*)=>*Addr*(*A*)**, if *A* is not live.

8. ***Addr*(*A*.*b*:*i*->*v*)=> *Addr*(*A*.*b*)**, if *A* is not live

9. ***Addr*(*A*[*i*]:*j*->*v*)=> *Addr*(*A*[*i*])**, if *A* is not live[2]

Figure 5-1: Reduction Rules in the absence of assignments

1. ***Addr*(let *definitions* in *value-expression* end)=> *Addr*(*value-expression*)**

2. if an assignment *A*:=*exp* is in a simple let-expression, then ***Addr*(*A*)=>*Addr*(*exp*)** unless the terminal expression of ***Addr*(*exp*)** is a name and it is live in the reordering chosen for the definitions and rest of the program graph.

3. if a parallel assignment [l_1,...,l_i,...]:=*exp* such that ***Addr*(*exp*) => [r_1,...,r_i,...]** occurs in a simple let-expression , then for each *i*, ***Addr*(l_i)=>*Addr*(r_i)** unless the terminal expression of ***Addr*(r_i)** is a name and it is live in the reordering chosen for the constructor and the rest of the program graph.

Figure 5-2: Rules for Assignments in simple let-expression

Interprocedural analysis is required to compute this information here and also in more complicated examples where non-locals are implemented by passing them as parameters. Proceeding with this information, we derive the following:

```
=>if n=0 then Addr(a) else Addr(a:n->x)
          assuming Addr(init(a,i,n))=>Addr(a)
=>if n=0 then Addr(a) else Addr(a) --since a is not live
=>Addr(a) --by rewriting conditional
```

Hence we can prove the consistency of the assumption that the function result can be shared with the first argument. This optimization which makes the result and the first argument share storage turns the value

[1] This is just one instance of a more general rule

[2] If a *new record* constructor is present in the language, two additional rules similar to the above two rules may be included

parameter into a var parameter and at the same time converts the function into a procedure. Before this optimization can be implemented, we need to check that the array *a* is not live in the callee after its transmission as a parameter.

To illustrate the rules for assignment in simple let-expression, consider the following fragment from the *puzzle* program. Constructs that have not been considered so far have been rewritten using functions and some simplifications also have been effected.

```
    function place (i: pieceType; j: position;
            puzzle: puztype; pieceCount: pctype) : placetype =
        let
            temp:integer:=class[i]
            plim: position  := pieceMax[i];
            puz: puztype  := letFunction(puzzle,plim,i,j)
            pc: pctype  := pieceCount: temp -> pieceCount[temp]-1;
            result: position  := leastFunction(j, size, puz)
        in [result, puz, pc]
        end
```

Assume it has already been discovered that ***Addr***(*letFunction*(*x*, *p*, *i*, *j*)) => ***Addr***(*x*) and ***Addr***(*leastFunction*(*j*, *k*, *x*)) => ***Addr***(*j*). We would like to target the assignments and also find the address expression of the function *place*. The first two definitions do not cause any interesting storage sharing to happen. The third definition can be targeted so that ***Addr***(*puz*)=>***Addr***(*puzzle*) since *puzzle* is not live in the function after this use. No reordering is needed since this is the only use of *puzzle*. The new-array can be evaluated in-place since *pieceCount* is not live. Hence, ***Addr***(*pc*)=>***Addr***(*pieceCount*). Finally, ***Addr***(*result*)=> ***Addr***(*j*) from the address expression for *leastFunction*, since *j* is not live in the function if the assignment for *puz* is evaluated before the assignment for *result*. The address expression for the function *place* is ***Addr***([*result*, *puz*, *pc*]) which can be reduced to [***Addr***(*j*), ***Addr***(*puzzle*), ***Addr***(*pieceCount*)]. Notice that all of them involve formal parameters of the function *place* and this signifies that the function *place* returns as result some updated version of these formal parameters. If copies are to be eliminated, these parameters can be changed into var parameters once it has been shown that the actual parameters of the function *place* are not live in any invocation. It might be advantageous to restrict conversion into var paramaters to structured values only to lessen some of the implementation difficulties encountered when this is attempted for scalars.

6. Fixpoint computation using reduction rules

To extend the method of computing address expressions using fixpoint iteration developed for the simple language in Section 4 to SAL, we need some modifications in the semantics. The most important is the presence of names other than parameters, both local and non-local.

6.1. Extensions

The new domain *AddrExp* is given by $\{\perp, \top\} \cup \mathcal{P}(X \cup Y) \cup \mathcal{P}(A \cup B)$ where

$Y = \{y \mid y$ is a local name$\}$

$Y = \{y \mid y$ is a local name$\}$

$B = \{y[l..m] \mid y \in Y;\ y$ is an array and $l,m \in RN_y\} - Y$

The operators μ and ν have to be extended with the following cases:

$\mu(a,b,g)$

$= g\ \exists\,(aa \subseteq a\ \&\ bb \subseteq b)\ \&\ aa \in \mathcal{P}(Y)$ and $bb \in \mathcal{P}(B)$ & vice-versa

$= g\ \exists\,(aa \subseteq a\ \&\ bb \subseteq b)\ \&\ aa \in \mathcal{P}(Y)$ and $bb \in \mathcal{P}(A)$ & vice-versa

$= g\ \exists\,(aa \subseteq a\ \&\ bb \subseteq b)\ \&\ aa \in \mathcal{P}(X)\ \&\ bb \in \mathcal{P}(B)$ & vice-versa

$= \mu(Addr(a), Addr(b), g)$ if a or $b \in \mathcal{P}(Y) \cup \mathcal{P}(B)$ & $Addr(a)$ or $Addr(b)$ can be reduced

$= [\mu(p_i, q_i, g_i) \mid i{:}1..n]$ if $a = [p_i \mid i = 1..n]$, $b = [q_i \mid i = 1..n]$ & $g = [g_i \mid i = 1..n]$

$\nu(a,b,g)$

$= \nu(Addr(a), Addr(b), g)$ if a or $b \in \mathcal{P}(Y) \cup \mathcal{P}(B)$ & $Addr(a)$ or $Addr(b)$ can be reduced

Since μ is associative, we will also use it for arbitrary number of arguments. The domain *AddrExp* still remains a cpo and ensures that the fixpoint computation always terminates.

Define ***Results*** of an expression as follows:

```
Results(let definitions in value-expression end)=Results(value-expression)

Results(let initial definitions
        while/for cond do
         redefinitions
        giving value-expression
        end)=Results(value-expression)

Results(if cond then c else a)= ∪( Results(c), Results(a))

For other expressions, Results(expression)={expression}
```

The ***Results*** of a function body returns all the syntactic expressions that are embedded in let-expressions or conditionals which may be returned as the value of the function.

6.2. Algorithm

The first part consists of the propagation of address expressions by fixpoint iteration. The address expression of each function is first set to $\perp$. For each function (in a topological order, if possible), compute μ of all the members of the set ***Results***(*f.body*) mapped by ***Addr***. Set the ***Addr***(*f*) equal to this value and propagate this information at all the sites of the function call by provisionally sharing all the names that are in the same subsets in the address expressions. It has to be provisional since additional information might cause some set of sharings to be inconsistent which then have to be undone. Repeat until there are no changes in the ***Addr*** expression computed for each function.

Next, all the actuals corresponding to the parameters that will be updated have to be checked for non-liveness. Finally, the provisional sharing between results and paramters made in the previous stages are

```
Let fList be the list of functions.
Set the address expression of each function f.Addr to $\perp$.
Compute a topological ordering for fList if possible[3] .

change:=true
while change do
 change:=false
 for each function $f \in$ fList in the topological order (if possible) do
     mapEx:=$\mu(\{Addr(results_i)| i=1..n \ \&\ results_i \in Results(f.body)\})$
     diff :=mapEx-f.Addr
     if diff $\neq$ nil then
       change:=true
       f.Addr:=mapEx
       Propagate this new value  at all the call sites of f
 endfor
endwhile
```

Figure 6-1: Algorithm for computing address expressions of functions

made final using the information obtained.

6.3. An example: counting permutations

To illustrate an example of fixpoint computation for address expressions, we present the function *permute* which counts the number of permutations of the elements of an array (Figure 6-2). The ***Results*** of the function body of *permute* is given by { [*A2*, *pctr2* + 1] , [*A*, *ctr* + 1] }. Mapping ***Results*** by ***Addr*** and computing *mapEx* gives the address expression as $[\{A\} \cup \{A1\}, \{ctr\} \cup \{pctr\}]$. Here we have used the information that $A2 \Rightarrow A1$ and $pctr2 \Rightarrow pctr$. This information has to be propagated provisionally at all the sites of the function call: hence *A1* shares storage with *A* and *pctr* with *ctr*. Similarly *A3* shares storage with *A1*(through **old** *A1*). Furthermore, *A2* shares storage with *A1*, *pctr* with *pctr2*. If we now compute the ***Addr*** expression of the function body of *permute*, all the ***Results*** get reduced to $[\{A\}, \{ctr\}]$. We now have found the fixpoint for the address expression of the function *permute*.

6.4. Divide and conquer problems

A very important subcase of interprocedural analysis occurs in divide and conquer algorithms. Such algorithms have significant opportunity for parallelism, but straightforward implementations may produce a significant number of copies, thus losing any performance advantage.

Consider a schema for divide and conquer problems using arrays as the data structure and assuming that the results can be combined by simple catenation (using the function **cat**):

[3] If there are irreducible cycles, any ordering would do but convergence might take longer

```
type permType=record X:arr; Y:integer end

function permute( n:integer;A:arr; ctr:integer) :permType=
 if n <> 1 then
 let
  A2:arr; pctr2:integer;
   [A2, pctr2] :=
    let
        A1:arr;pctr:integer;
        [A1, pctr] := permute(n-1, A, ctr);
        k:integer := n-1
    while k >= 1 do
        k := old k - 1;
        A3:arr;
        [A3, pctr] := permute(n-1,swap(old A1,n,old k), old pctr);
        A1 := swap(A3, n, old k)
    giving [A1, pctr]
    end
 in [A2, pctr2 + 1]
 end
 else [A, ctr + 1]
end;
```

Figure 6-2: Program to compute the number of permutations of an array

```
f(A:array[1..l] of T):=
 if l=1 then h(A)
 else cat(f(for i:1..l' be A[i]), f(for i:1..l-l' be A[l'+i]))
```

Since the base recursion involves the single element of the array, the function body can be evaluated without using more than $O(l)$ space, if it can be proved that there is no overlap in the parameters passed to the recursive calls of f. That is, there is possibility of sharing amongst the parameters among successive calls to f, if we show that there is no overlap in the elements accessed by the array former in the argument. Realization that this operation can be done in place leads to a very efficient program since the non-optimized program has to allocate and deallocate arrays at each level of the recursion in addition to the copy of the result from one level of the recursion to the next. Furthermore, if we assume that **cat** does not allocate storage or perform a copy if the objects are already adjacent, then we can eliminate the temporary storage and copying on return.

Given a set of recursive and non-recursive functions which implement a divide and conquer algorithm, we need to find out whether in-place modification of the data-structure is possible and safe. To do this, we need to find out how input data-structure is subdivided and how the result is composed. Computing the address expression for each function definition gives exactly this information. These address expressions describe how the address expression of a function result is defined in terms of the address expressions of its arguments. The fixpoints for the addresses of the functions are computed by iteration and by use of the reducing rules to simplify the expressions of the function bodies. However, type parameters make it possible to create type uncheckable expressions by using non-terminating function applications and looping constructs when defining index ranges for arrays. This may prevent address expressions to be computed. If index ranges and indices are assumed to be of a restricted syntactic form(for example, linear induction variables) for which symbolic analysis is helpful in determining simple algebraic properties and identities, the address expressions may be computable. This process is most easily understood from the viewpoint of an example (Figure 6-3).

```
--n is a power of 2
type arr(n:integer)= array[1..n] of integer;

function reverse(X:arr(n)):arr(n)=
 if n=1 then X
 else
  let A:arr(n):=X
  for i:1..n/2 do
   A:=(old A:i->old A[n-i+1]):n-i+1 ->old A[i]
  giving A
  end;

function sortbitonic(X:arr(n)):arr(n)=
 if n=1 then X
 else
  let A:arr(n):=X
  for i:1..n/2 do
   A:=if old A[i] < old A[i+n/2] then old A
      else (old A:i->old A[i+n/2]):
                 (i+n/2)->old A[i]
  giving
   let
    lower:arr(n/2):= for i:1..n/2 be A[i];
    upper:arr(n/2):= for i:1..n/2 be A[i+n/2];
   in cat(sortbitonic(lower), sortbitonic(upper))
   end
  end

function merge(X:arr(n);Y:arr(m)):arr(n+m)=
         sortbitonic(cat(X, reverse(Y)));

function sort(X:arr(n)):arr(n)=
 if n=1 then X
 else
  let
   lower:arr(n/2):= for i:1..n/2 be X[i];
   upper:arr(n/2):= for i:1..n/2 be X[i+n/2];
  in merge(sort(lower), sort(upper))
 end
```

Figure 6-3: Bitonic sort

Consider the function *sort*. It is clear that X can be sorted in place without any additional array storage or any copies. Determining that this is so in a compiler is a non-trivial matter. This depends on detecting the following in the process of proving consistency.

1. *lower* and *upper* are nonoverlapping (i.e., 1..$n/2$ and $n/2+1..n$ are non-overlapping)

2. the parameter X and the result of the function *sort* can share the same storage.

The last condition is the most complex. We show the initial equations that are written for the address expressions, and the simplification of the equations. Note that we require other optimization steps to occur to achieve the desired results. These steps are noted.

reverse
=>$\lambda X.\ X \cup$ *Addr*(redefining let-expression)
=>$\lambda X.\ X \cup A$
=>$\lambda X.\ X \cup X$
=>$\lambda X.\ X$

Hence *reverse*(X) can be done without using extra arrays. Now, consider the function *sortbitonic* -- abbreviated as *sb*.

sb
=>$\lambda X.X \cup$ *Addr*(redefining let-expression)
=>$\lambda X.X \cup$ *Addr*(simple let-expression)
=>$\lambda X.X \cup$ *Addr*(cat(*sb*(*lower*), *sb*(*upper*)))
=>$\lambda X.X \cup \nu(Addr(sb(lower)),\ Addr(sb(upper)),g_{lab_{\text{cat}}})$
=>$\lambda X.X \cup \nu(sb(A[1..n/2]),\ sb(A[n/2+1..n]),g_{lab_{\text{cat}}})$
=>$\lambda X.X \cup \nu(sb(X[1..n/2]),\ sb(X[n/2+1..n]),g_{lab_{\text{cat}}})$
--coalescing of A & X

The fixpoint iteration is as follows:
$sb_0 := \lambda X.\perp$
$sb_1 := \lambda X.X \cup \perp := \lambda X.X$
$sb_2 := \lambda X.X \cup \nu(X[1..n/2],\ X[n/2+1..n], g_{lab_{\text{cat}}}) := X$

Hence, ***Addr***(X) is a fixpoint solution for ***Addr***(*sb*(X)), and similarly, ***Addr***(X) is a fixpoint solution for ***Addr***(*sort*(X))

From the above analysis, we can conclude that *sort*(X) can be accomplished in place. Note that this process is both subtle and easily negated: if the function *reverse* is defined in a different way, say:
for i:1..n be $X[n-i+1]$
then an additional array is needed and one cannot conclude that *sort* can be done in-place.

7. Experimental results

A compiler for a substantial part of SAL has been implemented to verify the effectiveness of the approach. This compiler generates an intermediate code called UCODE and has many other aspects we have not discussed. They are fully covered in a thesis [7].

The timings for the following programs on a MicroVax-II (without counting the output times except when negligible) were collected using the UNIX *time* command.

- **Bubblesort**: sorts an array of 1000 elements.
- **Life program**: 500 iterations on a board 10 by 10 (with border 12 by 12)
- **Matrix multiply**: of two 100 by 100 integer matrices
- **8 queens**: Finds all the 92 solutions.
- **puzzle**: finds the solution. This is a very highly recursive and computationally demanding program for solving a three-dimensional puzzle. Often used for benchmarking C and other languages on workstations.
- **quicksort**: sorts the same array as the bubblesort does(1000 elements)
- **bitonic sort**: sorts an array of 1024 elements(has to be a power of 2)
- **perm**: counts the number of permutations of an array of 7 elements. This is iterated 5 times.
- **cyk**: the Cocke-Younger-Kasami algorithm parses an input string of 128 a's for the following ambiguous grammar:

  $A \rightarrow a$

  $A \rightarrow A\,A$
- **SIMPLE**: transliteration of the NEWRZ program, used in hydrodynamic calculations, considered by Ellis [6]

The user execution times(without I/O) in seconds are given in Table 7-1. The various optimization levels are as follows:

- No Opt: No optimization was done
- Opt1: All optimizations with no rangecheck elimination.
- Opt2: All optimizations with rangecheck elimination by analysis.
- Opt3: All optimizations with rangechecking turned off
- Opt4: All optimizations plus some very simple optimization on UCODE generated by inspection of code (peep-hole optimization in all the examples and 1 invariant moved in *puzzle* and *matrix multiply*)
- pc -O: Execution time for Berkeley Pascal at its highest optimization level

	No Opt	Opt1	Opt2	Opt3	Opt4	pc -O	%time
Bsort	1913.2	26.6	17.5	17.5	5.8(*)	14.9	39%
Life	23.4	22.6	18.4	18.4	14.5	8.2	177%
mm	72.6	82.2	48.2	48.2	27.7	32.3	86%
8	1.3	1.3	1.2	1.0	0.8	3.4	23%
simple	19.9	1.6	1.6	1.2	1.0	1.1	91%
cyk	58.0	56.9	41.9	39.0	30.1	16.1	187%
puzz	393.6	32.6	30.7	24.0	18.8	15.8(+)	119%
quick	12.5	2.8	2.8	1.5	1.2	1.3	92%
bitonic	14.3	2.9	2.9	2.5	2.2	1.6	138%
perm	5.5	3.5	3.5	2.5	2.3	2.5	92%

(*): Time with UOPT
(+): The Pascal version is faithful to the SAL version; if this is not attempted, we get a time of 13.3s

Table 7-1: Execution times of benchmarks in SAL and Berkeley Pascal.

We list also %time which is Opt4/pc -O.

It must be mentioned that UOPT [4], which is a UCode to UCode optimizer, could not be used except for *bubblesort*. Hence, there is substantial possibility for improvement in the execution times by use of register allocation, peephole optimization and other standard optimizations considered in compilers for imperative languages [1]. The execution times for *bubblesort* with and without UOPT are instructive. We believe that the timings could be improved by as much as 50% or more with an UCODE to UCODE optimizer.

To give another idea for the possibilities for improvement, we have optimised the UCODE generated by looking just for the simplest store-load peephole optimizations. *Opt4* in the table refers to peephole optimization done by inspection. In 2 cases (*puzzle, matrix multiply*) one invariant has been moved out of the loops. Even with all these handicaps, it is remarkable that we report execution times for six out of ten programs better than the best timings for Berkeley Pascal (pc -O). The program for *life* and *cyk* suffer because of the inability to define arrays partially in the SAL language.

8. Conclusions

It is very gratifying to see fairly theoretical approaches like abstract interpretation have such a direct bearing on critical issues like copy elimination.

References

[1] Aho, A. V., Ullman, J. D.
Principles of Compiler Design.
Addison-Wesley, 1977.

[2] Adrienne Bloss, Paul Hudak.
Variations on a strictness analysis.
In *ACM symposium on Lisp and Functional Programming.* ACM, Aug, 86.

[3] Celoni S.J., J.C., Hennessy, J.L.
SAL: A Single-Assignment Language for Parallel Algorithms.
Technical Report, Computer Systems Laboratory, Stanford University, July, 1981.

[4] Chow, F.
A Portable Machine-Independent Global Optimizer - Design and Measurements.
PhD thesis, Stanford University, December, 1983.

[5] Cousot and Cousot.
Abstract interpretation.
In *Annual ACM Symposium on Principles of Programming Languages.* ACM, Jan, 77.

[6] Ellis, J.R.
A Compiler for VLIW Architectures.
PhD thesis, Yale University, December, 1984.

[7] Gopinath, K.
Copy elimination in single assignment languages.
PhD thesis, Stanford University, June, 1987.

[8] Hudak and Young.
High-order strictness analysis in untyped lambda calculus.
In *Annual ACM Symposium on Principles of Programming Languages.* ACM, Jan, 1986.

[9] Paul Hudak.
A semantic model of reference counting and its abstraction .
In *ACM symposium on Lisp and Functional Programming.* ACM, Aug, 86.

[10] Alan Mycroft.
Abstract interpretation and Optimising transformations for applicative programs.
PhD thesis, Edinburgh University, 81.

[11] Jerald Schwarz.
Verifying the safe use of destructive operations in applicative programs.
In *Program transformations-- Proceedings of the 3rd Int'l Sym. on Programming.* 1978.

[12] Wadler.
Listlessness is better than Laziness:Lazy evaluation and Garbage collection at Compile time.
In *ACM symposium on Lisp and Functional Programming.* ACM, Aug, 1984.